PART 0
Beginnings

▶ **CHAPTER 1:** Primer

COPYRIGHTED MATERIAL

1

Primer

WHAT'S IN THIS CHAPTER?

- Understanding strategies
- Reviewing Lambda calculus
- Infering types
- Understanding mutability
- Creating your own bindings

Object-oriented programming has been with us for close to two decades, if not longer; its expressions of those concepts via the languages C# and Visual Basic, and the platform on which they run, the CLR, have been more recent, only since 2002. The community and ecosystem around the object-oriented paradigm is vast, and the various traps and pitfalls around an object-oriented way of thinking has seen copious discussion and experience, and where disagreements occur, reasoned and heated debate. From its inception more than 40 years ago, through its exploration in languages invented yesterday, an object-oriented approach to languages has received the benefit of the attention of some of the smartest language and tool designers in the industry, and a highly permutational approach to ancillary features around the language, such as garbage collection, strong-versus-weak typing, compilation-versus-interpretation, and various hybrids thereof, full-fidelity metadata, parameterized types, and more; no stone, it seems, remains unturned.

One of the principal goals of an object-oriented language is the establishment of user-defined types (UDTs) that not only serve to capture the abstractions around the users' domain, but also the capability to reuse those types in a variety of different scenarios within the same domain without modification. Sometimes this domain is a business-flavored one — at the start of the 21st century these kinds of types were called *business objects* and later *domain types*. Sometimes the domain is an infrastructural one, such as presentation or communication,

and these types are frequently collected together in a large bundle known as a *framework* or *class library*. And despite the relatively slow start that C++ class libraries saw during C++'s heyday, the combination of fast, cheap network access and volunteers willing to share their efforts has led to a proliferation of these infrastructural bundles that dwarfs anything the industry has seen before.

Unfortunately, despite the huge success of C++, Java, and .NET, the original premise of the object-oriented, that developers could take objects "off the shelf" and reuse them without modification, has yet to occur for anything but the infrastructural domain. Even in that domain, debate and duplication rages over subtle points of design that can only be changed by changing the underlying source code making up those infrastructural types.

More important, as the 20th century came to a close and the 21st loomed on the horizon, developers began to find "edges," limitations to the object-oriented paradigm that saw no easy answer. These edges pushed the object-oriented community to amend its approach slightly, placing those objects into *containers* and relying on the container to provide a set of services that could not easily be captured by object-oriented methodology. These *component containers*, exemplified by the Enterprise Java Beans standard (and its later lighter-weight successor, Spring) and COM+/EnterpriseServices found some traction but were never wholly adored by the developers who used them.

In time, those containers were amended to take a more coarse-grained approach, seeking a certain degree of simplification to enable for a more interoperable capacity, and these new domain bundles received a new name: *services*. Although Service-Oriented Architecture (SOA) sounded an entirely new style of programming, developers seeking to take their traditional object-oriented concepts into the service-oriented realm found themselves struggling with even the simplest of designs.

Despite the guidance of well-honed discussions around object-oriented design (called *design patterns* at first, then later just *patterns*), the more developers sought to model their software after the reality of the businesses around them, the more they struggled to achieve the reusability promised them. Coupled with this came the disquieting realization that object languages had not only failed to produce that set of "Tinkertoys" that developers or users could just reuse off the shelf, but also that some domains defied some of the modeling characteristics of object languages entirely — some domains were just complicated and awkward to model in objects.

Then, subtly, a new problem emerged: the underlying hardware ceased its steady march of improved performance and instead began to respond to the beat of a different drum, that of *multi-core*. Suddenly, where programmers used to face problems in a single- or slightly-multi-threaded environment, now the hardware demanded that additional performance would come only from greater and greater parallelization of code. Where the object-oriented developers of the 20th century could assume that only a single logical thread of execution would operate against their objects, the developers of the 21st century has to assume that many logical threads of execution will all be hammering on their objects simultaneously. The implication, that developers must now consider all possible interactions of multiple threads on every operation available on the types they define, continues to hang like a proverbial Sword of Damocles over the heads of object-oriented programmers even as the first decade of the new century came to a close.

This chapter attempts to demonstrate some of the technical challenges C# and Visual Basic developers (and, to a logical extent, their cousins and predecessors in Java and C++) have faced when using the object-oriented paradigm exclusively. In the beginning, challenges will be addressed with

programming techniques available to the developer using off-the-shelf tools and technology, but as the problems surmount, so will the desire to change those tools into something more, something that demonstrates the need for a new approach to programming beyond the traditional object-oriented one, and a new language to explore and expose those concepts more succinctly and directly.

SETUP

Imagine a system, a simple one (to start) for tracking students and instructors and the classes they teach. In keeping with traditional object-oriented thought, several domain types are defined, here in C# 2.0, though the actual language — C#, Visual Basic, C++/CLI (any .NET language would work) — in which they are defined makes little difference:

```
class Person
{
    private string m_first;
    private string m_last;
    private int m_age;

    public Person(string fn, string ln, int a)
    {
        FirstName = fn;
        LastName = ln;
        Age = a;
    }
    public string FirstName
    {
        get { return m_first; }
        set { m_first = value; }
    }
    public string LastName
    {
        get { return m_last; }
        set { m_last = value; }
    }
    public int Age
    {
        get { return m_age; }
        set { m_age = value; }
    }
}

class Student : Person
{
    private string m_major;

    public Student(string fn, string ln, int a, string maj)
        : base(fn, ln, a)
    {
        Major = maj;
    }
    public string Major
```

```
    {
        get { return m_major; }
        set { m_major = value; }
    }
}

class Instructor : Person
{
    private string m_dept;

    public Instructor(string fn, string ln, int a, string dept)
        : base(fn, ln, a)
    {
        Department = dept;
    }
    public string Department
    {
        get { return m_dept; }
        set { m_dept = value; }
    }
}

class Class
{
    private string m_name;
    private List<Student> m_students = new List<Student>();
    private Instructor m_instructor;

    public Class(string n)
    {
        Name = n;
    }
    public string Name
    {
        get { return m_name; }
        set { m_name = value; }
    }
    public Instructor Instructor
    {
        get { return m_instructor; }
        set { m_instructor = value; }
    }
    public List<Student> Students
    {
        get { return m_students; }
    }
}
```

As with most systems of its type, the system begins simply: there are two kinds of "Person"s in the system, `Students` and `Instructors`, and `Classes` are taught to `Students` by `Instructors`. So, building up from basic parts, lists of `Instructors` and `Students` might look like this:

```
class Program
{
    static List<Student> Students = new List<Student>();
    static List<Instructor> Instructors = new List<Instructor>();
```

```
        static Program()
        {
            Instructors.Add(new Instructor("Al", "Scherer", 38,
                            "Computer Science"));
            Instructors.Add(new Instructor("Albert", "Einstein", 50,
                            "Physics"));
            Instructors.Add(new Instructor("Sigmund", "Freud", 50,
                            "Psychology"));
            Instructors.Add(new Instructor("Aaron", "Erickson", 35,
                            "Underwater Basketweaving"));

            Students.Add(new Student("Matthew", "Neward", 10,
                         "Grade school"));
            Students.Add(new Student("Michael", "Neward", 16,
                         "Video game design"));
            Students.Add(new Student("Charlotte", "Neward", 38,
                         "Psychology"));
            Students.Add(new Student("Ted", "Neward", 39,
                         "Computer Science"));
        }
    }
```

Obviously, in a real-world program, these lists of objects will be stored in some form of long-term storage, such as a relational database, but this suffices for now.

IT'S THAT TIME OF YEAR AGAIN...

At the beginning of the school year, new classes are created and entered into the system — again, this is modeled in the example as a simple list:

```
List<Class> classesFor2010 = new List<Class>();
classesFor2010.Add(new Class("Scala for .NET Developers"));
classesFor2010.Add(new Class("F# for .NET Developers"));
classesFor2010.Add(new Class(
                   "How to play pranks on teachers"));
classesFor2010.Add(new Class(
                   "Baskets of the Lower Amazon"));
classesFor2010.Add(new Class("Child Psych"));
classesFor2010.Add(new Class("Geek Psych"));
```

And when the classes have been set up (`Instructors` will be assigned later, after the `Instructors` have determined who is lowest on the totem pole and has to actually teach this year), the students need to log in to the system and register for classes:

```
Console.Write("Please enter your first name:");
string first = Console.ReadLine();
Console.Write("\nPlease enter your last name:");
string last = Console.ReadLine();
```

After the student has entered this information, the two strings must somehow be reconciled into a `Student` object, a process that usually involves searching the list:

```
foreach (Student s in Students)
{
```

```
            if (s.FirstName == first && s.LastName == last)
            {
                // ... Do something
            }
        }
```

This feels sloppy somehow — after some deeper thought, several things emerge as "wrong."

First, an obvious bottleneck emerges if this list becomes large; for a school of a half-dozen `Instructors` and a few dozen `Students`, this simple one-at-a-time comparison works well enough, but if the system grows to incorporate campuses all across the world and millions of `Students`, this is going to break down quickly.

Second, when it comes time to select an `Instructor` for a class, similar kinds of search needs to happen, and simply repeating the foreach loops over and over again is a violation of the DRY (Don't Repeat Yourself) principle — there's no reusability. This problem will only magnify itself if the searches are somehow optimized, because the optimizations will need to happen for each search.

It may seem tempting to brush these concerns off as irrelevant, because if the data is stored in a relational database, the performance and scalability concerns become issues of SQL and database tuning. Stay with the point for now, and trust in that this is merely the tip of the iceberg.

The obvious object-oriented solution to the problem, then, is to create custom data structures for storing the `Students` and `Instructors`:

```
class InstructorDatabase
{
    private List<Instructor> data = new List<Instructor>();
    public InstructorDatabase() { }

    public void Add(Instructor i) { data.Add(i); }
    public Instructor Find(string first, string last)
    {
        foreach (Instructor i in data)
        {
            if (i.FirstName == first && i.LastName == last)
                return i;
        }
        return null;
    }
}

class StudentDatabase
{
    private List<Student> data = new List<Student>();
    public StudentDatabase() { }

    public void Add(Student i) { data.Add(i); }
```

```
    public Student Find(string first, string last)
    {
        foreach (Student i in data)
        {
            if (i.FirstName == first && i.LastName == last)
                return i;
        }
        return null;
    }
}
```

At first glance, this seems like a good idea, but a longer look reveals that these two classes differ in exactly one thing — the kind of data they "wrap." In the first case, it's a list of `Instructors`, and in the second, a list of `Students`.

The sharp C# 2.0 programmer immediately shouts out, "Generics!" and after enduring the quizzical looks of the people sharing the room, looks to create a single type out of them, like this:

```
class Database<T>
{
    private List<T> data = new List<T>();
    public Database() { }

    public void Add(T i) { data.Add(i); }
    public T Find(string first, string last)
    {
        foreach (T i in data)
        {
            if (i.FirstName == first && i.LastName == last)
                return i;
        }
        return null;
    }
}
```

but unfortunately, the C# compiler will balk at the code in `Find()`, because the generic type `T` can't promise to have properties by the name of `FirstName` and `LastName`. This can be solved by adding a type constraint in the generic declaration to ensure that the type passed in to the `Database` is always something that inherits from `Person`, and thus has the `FirstName` and `LastName` properties, like this:

```
class Database<T> where T: Person
{
    private List<T> data = new List<T>();
    public Database() { }

    public void Add(T i) { data.Add(i); }
    public T Find(string first, string last)
    {
        foreach (T i in data)
        {
            if (i.FirstName == first && i.LastName == last)
                return i;
        }
```

```
            return null;
        }
    }
```

which works, for now. Unfortunately, although this solves the immediate problem, what happens when the `Database` needs to search for a `Student` by major, or an `Instructor` by field? Because those are properties not specified on the `Person` type, once again the `Database` class will fail.

What's actually needed here is the ability to search via some completely arbitrary criteria, specified at the time the search is to happen — if this search were being done in SQL, the programmer could pass in a `WHERE` clause, a search predicate, by which the database could evaluate all the potential matches and return only those that met the criteria.

This sounds like a pretty good idea and one to which object-orientation has an answer: the Strategy pattern.

STRATEGY

In design patterns parlance, a Strategy is a setup where an object that implements an algorithm can be passed in for execution without the client knowing the actual details of the algorithm. More important, an appropriate Strategy can be selected at runtime, rather than being decided (some might say hard-coded) at compile-time. This is almost spot-on to what's needed here, except in this case the "algorithm" being varied is the criteria by which each potential match is evaluated.

In the classic Strategy implementation, an interface defines the parameters and result type to the algorithm:

```
interface ISearchCriteria<T>
{
    bool Match(T candidate);
}
```

This, then, enables the `Database` to be written to be entirely ignorant of the criteria by which to search:

```
class Database<T> where T : class
{
    private List<T> data = new List<T>();

    public T Find(ISearchCriteria<T> algorithm)
    {
        foreach (T i in data)
        {
            if (algorithm.Match(i))
                return i;
        }
        return null;
    }
}
```

The type constraint is still necessary, because the `Database` needs to return "null" in the event that the search fails. But now at least the `Database` is once again generic. Unfortunately, using it leaves something to be desired:

```
class Program
{
    static Database<Student> Students = new Database<Student>();
    static Database<Instructor> Instructors =
        new Database<Instructor>();

    class SearchStudentsByName : ISearchCriteria<Student>
    {
        private string first;
        private string last;
        public SearchStudentsByName(string f, string l)
        {
            first = f;
            last = l;
        }
        public bool Match(Student candidate)
        {
            return candidate.FirstName == first &&
                candidate.LastName == last;
        }
    }

    static void Main(string[] args)
    {
        // ...
        Student s = null;
        while (s == null)
        {
            Console.Write("Please enter your first name:");
            string first = Console.ReadLine();
            Console.Write("\nPlease enter your last name:");
            string last = Console.ReadLine();
            s = Students.Find(
                new SearchStudentsByName(first, last));
            if (s == null)
                Console.WriteLine("Sorry! Couldn't find you");
        }
        // Do something with s
```

Yikes. The code definitely got a little bit easier to use at the point of doing the search, but now a new class has to be written every time a different kind of search needs to happen, and that class will have to be accessible every place a search could be written.

Fortunately, the savvy C# 2.0 developer knows about delegates and their extremely powerful cousins, anonymous methods. (Equally as fortunate, the savvy C# 2.0 developer knows not to shout things out in a room full of people while reading a book.)

THE DELEGATE STRATEGY

The whole interface-based Strategy approach can be eliminated in favor of a well-defined delegate type and an instance of an anonymous method:

```
delegate bool SearchProc<T>(T candidate);
class Database<T> where T : class
{
    private List<T> data = new List<T>();
    public Database() { }

    public void Add(T i) { data.Add(i); }
    public T Find(SearchProc<T> algorithm)
    {
        foreach (T i in data)
        {
            if (algorithm(i))
                return i;
        }
        return null;
    }
}
```

The real savings comes at the point where the student login code does the lookup; because the search now takes a delegate instance, the criteria by which the `Student` is looked up can be as rich or as simple as the case demands:

```
Student s = null;
while (s == null)
{
    Console.Write("Please enter your first name:");
    string first = Console.ReadLine();
    Console.Write("\nPlease enter your last name:");
    string last = Console.ReadLine();
    s = Students.Find(delegate (Student c) {
                          return
                              c.FirstName == first &&
                              c.LastName == last
                      });
    if (s == null)
        Console.WriteLine("Sorry! Couldn't find you");
}
// Do something with s
```

Now, all kinds of performance optimization can be done in the `Database<T>` class, because the client code remains ignorant of *how* the search is done. Instead, it simply specifies *what* to match (or if you will, how the *match* is done, instead of how the *search* is done).

However, the `Database` isn't done yet: if the `Database` is later going to find all Instructors that teach a particular subject, it needs the capability to return more than one object if the criteria is matched:

```
class Database<T> where T : class
{
```

```
    private List<T> data = new List<T>();

    // ...

    public T[] FindAll(SearchProc<T> algorithm)
    {
        List<T> results = new List<T>();
        foreach (T i in data)
        {
            if (algorithm(i))
                results.Add(i);
        }
        return results.ToArray();
    }
}
```

C# 2.0 saw much of this coming and predefined those delegate types already as part of the FCL: the `Predicate<T>` and `Func<T>` delegate types, the first used to yield a bool result (like the `SearchProc` previously defined) and the other used to simply "act" upon the value passed in (such as printing it to the console or something similar). In the spirit of "Code not written or removed means code not written with bugs, or maintained so that bugs are introduced later," this means that the code can be refactored to remove the redundant delegate type declaration and use those already defined:

```
class Database<T> where T : class
{
    private List<T> data = new List<T>();
    public Database() { }

    public void Add(T i) { data.Add(i); }
    public T Find(Predicate<T> algorithm)
    {
        foreach (T it in data)
            if (algorithm(it))
                return it;
        return null;
    }
    public T[] FindAll(Predicate<T> algorithm)
    {
        List<T> results = new List<T>();
        foreach (T it in data)
            if (algorithm(it))
                results.Add(it);
        return results.ToArray();
    }
}
```

Other kinds of operations on the `Database` not yet implemented (but should be) quickly come to mind, such as taking some kind of action on each of those returned objects. To be precise, three kinds of operations should be supported on `Database<T>`: `Filter`, `Map`, and `Reduce`:

```
delegate U Accumulator<T, U>(T src, U rslt);
class Database<T> where T : class
{
    private List<T> data = new List<T>();
```

```
        public Database() { }

        public void Add(T i) { data.Add(i); }

        public IEnumerable<T> Filter(Predicate<T> pred)
        {
            List<T> results = new List<T>();
            foreach (T it in data)
                if (pred(it))
                    results.Add(it);
            return results;
        }
        public IEnumerable<U> Map<U>(Func<T, U> transform)
        {
            List<U> results = new List<U>();
            foreach (T it in data)
                results.Add(transform(it));
            return results;
        }
        public U Reduce<U>(U startValue, Accumulator<T, U> accum)
        {
            U result = startValue;
            foreach (T it in data)
                result = accum(it, result);
            return result;
        }

        public T Find(Predicate<T> algorithm)
        {
            return Filter(algorithm).GetEnumerator().Current;
        }
        public T[] FindAll(Predicate<T> algorithm)
        {
            return new List<T>(Filter(algorithm)).ToArray();
        }
    }
```

When those three operations are in place, any other operation can be defined in terms of those three through various combinations of them.

By defining their parameters in terms of `IEnumerable<T>`, instead of as a raw array as the earlier definitions did, any sort of `IEnumerable<T>` could be used, including lists, arrays, or even the anonymous iterator defined using the C# 2.0 yield return statement:

```
        public IEnumerable<T> Filter2(Predicate<T> pred)
        {
            foreach (T it in data)
                if (pred(it))
                    yield return it;
        }
        public IEnumerable<U> Map2<U>(Func<T, U> transform)
```

```
{
    foreach (T it in data)
        yield return (transform(it));
}
```

In terms of their purpose, `Filter` is the easiest to grasp — it applies the `Predicate` to each element in the `Database` and includes that element if the `Predicate` returns true.

`Map` is less obvious — it applies a `Func` (an operation) to each element in the `Database`, transforming it into something else, usually (though not always) of a different type. If, for example, the system needs to extract a list of all the `Students`' ages, `Map` can transform the `Student` into an age:

```
foreach (int a in
    Students.Map(delegate(Student it)
    { return it.Age; }))
{
    Console.WriteLine(a);
}
```

The last, `Reduce`, is the most complicated, largely because it is the most fundamental — both `Map` and `Filter` could be rewritten to use `Reduce`. `Reduce` takes the collection, a delegate that knows how to extract a single bit of information from the element and perform some operation on it to yield a rolling result and hand that back. The easiest thing to do with `Reduce` is obtain a count of all the elements in the collection, by incrementing that accumulator value each time:

```
int count =
    Students.Reduce(0, delegate(Student st, int acc)
                            {
                                return acc++;
                            });
```

Or if for some reason the total of all the `Students`' ages added together were important, `Reduce` can produce it by adding the age to the accumulator each time through:

```
int sumAges =
    Students.Reduce(0, delegate(Student st, int acc)
                            {
                                return st.Age + acc;
                            });
```

In truth, this is useless information — what would be much more interesting is the average of all the `Students`' ages, but this is a bit trickier to do with Reduce — because the average is defined as the total divided by the number of elements in the collection, Reduce has to be used twice:

```
float averageAge =
    (Students.Reduce(0, delegate(Student st, float acc)
                            {
                                return st.Age + acc;
                            }))
    /
    (Students.Reduce(0, delegate(Student st, float acc)
```

```
                                        {
                                            return acc + 1;
                                        }));
```

But even more intriguingly, a collection of `Students` can be "reduced" to an XML representation by applying the same approach and transforming a `Student` into a `string` representation:

```
string studentXML =
    (Students.Reduce("<students>",
                     delegate(Student st, string acc)
                     {
                         return acc +
                             "<student>" +
                             st.FirstName +
                             "</student>";
                     })) + "</students>";
```

If some of this sounds familiar, it is because much of this was later expanded to be a major part of the C# 3.0 release. LINQ, Language-Integrated Query, centers on these same core principles. Using C# 3.0's capability to define new methods from "outside" a class (extension methods), C# 3.0 defined a series of these methods directly on the collection classes found in the .NET Framework Class Library, thus making the `Database` type even simpler:

```
class Database<T> where T : class
{
    private List<T> data = new List<T>();
    public Database() { }

    public void Add(T i) { data.Add(i); }
    public T Find(Predicate<T> algorithm)
    {
        return data.Find(algorithm);
    }
    public T[] FindAll(Predicate<T> algorithm)
    {
        return data.FindAll(algorithm).ToArray();
    }
}
```

And, of course, if the `List<T>` holding the `Student` objects is available for public consumption, perhaps via a property named `AsQueryable`, as is the convention in LINQ, the Students' ages can be counted, summed, and averaged using a LINQ expression:

```
count =
    Students.AsQuerayable.Aggregate(0, (acc, st) => ++acc);
sumAges =
    Students.AsQuerayable.Aggregate(0,
        (acc, st) => st.Age + acc);
averageAge =
    Students.AsQuerayable.Aggregate(0.0F,
        (acc, st) => ++acc)
    /
    Students.AsQuerayable.Aggregate(0.0F,
        (acc, st) => st.Age + acc);
```

As can be surmised from the preceding code, the LINQ `Aggregate` extension method is the moral equivalent of the `Reduce` written earlier. And as was demonstrated, this means LINQ can be used to "reduce" a collection of Students into an XML representation.

C# 3.0 also offered a slightly more terse way of specifying those delegates to be passed in to the `Database`, something called a *lambda expression*:

```
Student s = null;
while (s == null)
{
    Console.Write("Please enter your first name:");
    string first = Console.ReadLine();
    Console.Write("\nPlease enter your last name:");
    string last = Console.ReadLine();
    s = Students.Find(c => c.FirstName == first &&
                           c.LastName == last );
    if (s == null)
        Console.WriteLine("Sorry! Couldn't find you");
}
// Do something with s
```

The etymology of this name stems quite deeply in computer science history, to a mathematician named Loronzo Church, who discovered a subtle yet very powerful idea: if mathematical functions are thought of as things that can be passed around like parameters (just as delegates can), then all mathematical operations can be reduced to functions taking functions as parameters. And this body of work came to be known as the *lambda calculus*, after the Greek symbol that served as the placeholder symbol for the function name.

LAMBDA CALCULUS (BRIEFLY)

Without getting too deeply into the academic details, Church observed that if the actual function could be passed around, then various operations that normally seem distinct and individual could be collapsed together into a single *higher-order* function.

Consider the following two basic math operations and their C# equivalents:

```
static int Add(int x, int y) { return x + y;  }
static int Mult(int x, int y) { return x * y; }
static void MathExamples()
{
    int x = 2;
    int y = 3;
    int added = x + y;
    int multed = x * y;
    int addedagain = Add(x, y);
    int multedagain = Mult(x, y);
}
```

What Church realized is that if the actual operation — adding, multiplying, whatever — is abstracted away as a parameter, then both of these operations can be described using a single function:

```
delegate int BinaryOp(int lhs, int rhs);
static int Operate(int l, int r, BinaryOp op)
```

```
    {
        return op(l, r);
    }
    static void MathExamples()
    {
        int x = 2;
        int y = 3;
        // using explicit anonymous methods
        int added = Operate(x, y,
                            delegate(int l, int r){ return l+r; });
        int multd = Operate(x, y,
                            delegate(int l, int r){ return l*r; });
        // using lambda expressions
        int addedagain = Operate(x, y, (l, r) => l + r);
        int multdagain = Operate(x, y, (l, r) => l * r );
    }
```

When used this way, it doesn't make a lot of sense, because it would seem obvious to just write x + y, but now that the operation is abstracted away from the actual point of performing the operation, any kind of operation can be passed in.

This has some interesting implications. Consider, for example, an all-too-common business rule: `Classes` are associated with a given field of study (which must now appear on the `Class` object as another property, `Field`), because only certain kinds of `Students` can take certain kinds of `Classes` — for example, a `Student` studying `Computer Science` can take `Computer Science` classes, as can `Students` studying video game design, but `Computer Science` students are forbidden from taking anything that won't help them learn computer science better, such as `Underwater Basketweaving` classes. Meanwhile, `Video game design` is a pretty open-ended major and accepts just about anything except `Fashion Design` classes, whereas `Physics` majors will need to know some `Physics` and `Computer Science` but nothing else, and so on. (Like so many of its kind, this business rule made all kinds of sense back when it was first created, and nobody still with the company remembers why it was created in the first place, so don't question it now.) When a `Student` signs up for a `Class`, this business rule needs to be enforced, but where?

This kind of validation has historically plagued the object-oriented designer; effective enforcement of the rule requires knowledge coming from two different places, the `Student` and the `Class`. As such, it seems to defy logical placement: If the validation routine lives on the `Student`, the `Student` class then has to have awareness of every kind of `Class` in the system, a clear violation of separation of concerns, and vice versa if the validation routine lives on the `Class`.

If the validation is abstracted away, however, now the validation can occur without having to know the actual details yet:

```
delegate bool MajorApproval(Class cl);
class Student : Person
{
    // ...
    private MajorApproval m_majorApproval;

    // ...

    public MajorApproval CanTake
    {
```

```
        get { return m_majorApproval; }
    }
}

class Class
{
    // ...

    public bool Assign(Student s)
    {
        if (s.CanTake(this))
        {
            Students.Add(s);
            return true;
        }
        return false;
    }
}
```

Now, the validation code can be kept in a third place, thus localizing it to neither the `Student` nor the `Class`, but someplace accessible to either, such as a `ProgramSignup` collection someplace that the `Student` constructor can access (or the `Class.Assign` method could consult directly, depending on the developer's sense of aesthetics):

```
ProgramValidation["ComputerScience"] =
    c => c.Field == "Computer Science";
ProgramValidation["Physics"] =
    c => c.Field == "Computer Science" ||
         c.Field == "Physics";
ProgramValidation["Psychology"] =
    c => c.Field == "Psychology" ||
         c.Field == "Grade school";
ProgramValidation["Grade school"] =
    c => false;
ProgramValidation["Video game design"] =
    c => c.Field != "Underwater Basketweaving";
```

It should be noted that yes, something similar to this could be modeled in a traditional object-oriented way, usually by modeling the rules as function objects (like the Strategy approach earlier), also sometimes referred to as *functors*, and has been around for quite a while; C and C++ used pointers-to-functions to do things like this for decades.

The lambda calculus also permits more than just "lifting" the operation out and capturing it, though. As originally expressed, the lambda calculus also states that if a function can accept a function as a parameter, then all functions are essentially functions of one parameter, even though they may appear otherwise.

For example, returning to the basic mathematic operation of addition, the basic C# method of earlier:

```
static int Add(int x, int y) { return x+y; }
static void MoreMathExamples()
{
    int result = Add(2, 3);
}
```

In the lambda calculus, this operation can be thought of as asking a function to take an `int` and another function, where that second function takes an `int` and returns an `int` back. So, in C# 2.0, converting this to look like what the lambda calculus implies, starting with the `Add` method converted to a delegate operation:

```
private delegate int Operation(int l, int r);
static void MoreMathExamples()
{
    int result = Add(2, 3);

    Operation add = delegate(int l, int r) { return l + r; };
    int result2 = add(2, 3);
}
```

then means the delegate can be broken up into two different delegates — the first handing back a delegate that in turn knows how to do the actual operation:

```
delegate InnerOp DelegateOp(int r);
delegate int InnerOp(int l);
static void MoreMathExamples()
{
    int result = Add(2, 3);

    Operation add1 = delegate(int l, int r) { return l + r; };
    int result2 = add1(2, 3);

    DelegateOp add2 = delegate(int l)
                          {
                              return delegate(int r)
                                         {
                                             return l + r;
                                         };
                          };
    int result3 = add2(2)(3);
}
```

This process is known as *currying*, named after Haskell Curry, another mathematician who was famous (among mathematicians, at least). Because this sequence of steps should be applicable for more than just integers, a quick application of generics makes it available for any type:

```
Func<int,Func<int, int>> add4 =
    delegate(int l)
        {
            return delegate(int r)
                       {
                           return l + r;
                       };
        };
int result4 = add4(2)(3);
```

Then, on top of all this, the whole process can be further genericized by creating a standard-purpose method for doing it:

```
delegate U Op<T1, T2, U>(T1 arg1, T2 arg2);
delegate U Op<T1, U>(T1 arg1);
```

```
static Op<T1, Op<T2, U>> Curry<T1, T2, U>(Op<T1, T2, U> fn)
{
    return delegate(T1 arg1)
               {
                   return delegate(T2 arg2)
                              {
                                  return fn(arg1, arg2);
                              };
               };
}
static void MoreMathExamples()
{
    int result = Add(2, 3);

    Operation add2 = delegate(int l, int r) { return l + r; };
    int result2 = add2(2, 3);

    DelegateOp add3 = delegate(int l)
                          {
                              return delegate(int r)
                                         {
                                             return l + r;
                                         };
                          };
    int result3 = add3(2)(3);

    Func<int,Func<int, int>> add4 =
        delegate(int l)
            {
                return delegate(int r)
                         {
                             return l + r;
                         };
            };
    int result4 = add4(2)(3);

    Op<int, int, int> add5 =
        delegate(int l, int r) { return l+r; };
    int result5 = add5(2, 3);
    Op<int, Op<int, int>> curriedAdd = Curry(add5);
    int result6 = curriedAdd(2)(3);
}
```

It's horribly obtuse, and no sane C# developer would write add this way…unless they wanted to build up chains of functions calling functions in a highly generic way:

```
Op<int, int> increment = curriedAdd(1);
int result7 = increment(increment(increment(2)));
```

Although it seems awkward to think about at first, composing functions in this way means a new level of reusability has opened up, that of taking operations (methods) and breaking them into smaller pieces that can be put back together in new and interesting ways.

The most obvious use of this is to "pipeline" functions together in various ways, permitting reuse of behavior at a level previously unheard of in the object-oriented space. Functionality can now be

written in small chunks, even as small as simple operations and then composed together, such as what might be needed for input validation code for a web application. However, making this work in a syntax that doesn't drive the average C# developer insane (if it hasn't already) is difficult, which leads to the next question — what if we could somehow make the syntax cleaner and easier to read and understand?

TYPE INFERENCE

One thing is apparent from all this, particularly the definition of the `Curry` method: This is heavily genericized code, and it's not easy to read. It's a long way from `List<T>`! Fortunately, C# 3.0 introduced anonymous local variables, effectively informing the compiler that it is now responsible for determining the type of the declared local variable:

```
var add9 = Curry(add5);
int result9 = add9(2)(3);
```

Here, the compiler uses the context surrounding the variable declaration to figure out, at compile-time, precisely what the type of the local variable should be. In other words, despite the vague-sounding `var` syntax, this is still a full statically-typed variable declaration — it's just that the programmer didn't have to make the declaration explicit because the compiler could figure it out instead.

Theoretically, this means now that the compiler can remove responsibility for the "physical details" about the code from the programmers' shoulders:

```
var x = 2;
var y = 3;
var add10 = add9(x)(y);
```

Are x and y local variables, or are they property declarations? If these were members of a class, would they be fields or properties or even methods? Is `add10` a method or delegate? More important, does the programmer even care? (In all but a few scenarios, probably not.)

Ideally, this is something that could be layered throughout the language — such that fields, properties, method parameters, and more, and could all be inferred based on context and usage, such as:

```
// This is not legal C# 3.0
class Person
{
    public Person(var firstName, var lastName, var age)
    {
        FirstName = firstName;
        LastName = lastName;
        Age = age;
    }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
    public string FullName {
      get { return FirstName + " " + LastName; }
    }
}
```

C# 3.0 provides some of this, via the automatic property declaration — it assumes the task of creating a field and the get/set logic to return and assign to that field, respectively, but unfortunately, C# 3.0 will choke on the `var` declaration as a method parameter or as a field. And 3.0 allows only inference for local variable declarations — any attempt to use these things as fields in an object will require the explicit declaration and, potentially, all the disconcerting angle brackets.

Additionally, looking at the previous example, some of C#'s syntactic legacy begins to look awkward — specifically, the use of the `var` as a type prefix is somewhat redundant if the compiler is going to infer the type directly, so why continue to use it?

```
// This is not legal C# 3.0
class Person
{
    public Person(firstName, lastName, age)
    {
        FirstName = firstName;
        LastName = lastName;
        Age = age;
    }
    public FirstName { get; set; }
    public LastName { get; set; }
    public Age { get; set; }
    public FullName {
      get { return FirstName + " " + LastName; }
    }
}
```

Despite the compiler's best efforts, though, it may be necessary to provide the type as a way of avoiding ambiguity, such as when the compiler cannot infer the type or finds any number of potential inferences. `FirstName` and `LastName` can be assumed to be strings, since the `FullName` property adds them together against a `constant` string, something (presumably) only strings can do. `Age`, however, is an ambiguity: It could be just about any object type in the system, because it is never used in a context that enables the compiler to infer its numerical status. As a result, the compiler needs a small bit of help to get everything right.

If the type declaration prefix syntax has been thrown away, though, then something else will have to take its place, such as an optional type declaration suffix syntax:

```
// This is not legal C# 3.0
class Person
{
    public Person(firstName, lastName, age : int)
    {
        FirstName = firstName;
        LastName = lastName;
        Age = age;
    }
    public FirstName { get; set; }
    public LastName { get; set; }
    public Age { get; set; }
    public FullName {
      get { return FirstName + " " + LastName; }
    }
}
```

Of course, types need not be the only thing inferred by the compiler; because `public` is the most common access modifier for methods, constructors and properties, and `private` is most common for fields, let those be the inferred default:

```
// This is not legal C# 3.0
class Person
{
    Person(firstName, lastName, age : int)
    {
        FirstName = firstName;
        LastName = lastName;
        Age = age;
    }
    FirstName { get; set; }
    LastName { get; set; }
    Age { get; set; }
    FullName {
      get { return FirstName + " " + LastName; }
    }
}
```

While syntax is under the microscope, the constructor syntax is a bit strange — why is repeating the type's name necessary? And because most types have a principal constructor to which all other constructors defer, if it even has multiple constructors at all, that constructor should have a more prominent place in the type's declaration:

```
// This is not legal C# 3.0
class Person(firstName, lastName, age : int)
{
    FirstName { get; set; }
    LastName { get; set; }
    Age { get; set; }
    FullName {
      get { return FirstName + " " + LastName; }
    }
}
```

Problem is, the constructor body is now missing, and the assignment of the constructor parameters to the respective properties is lost, unless somehow the body of the class can serve as the body of the constructor, and the property declaration can know how to "line up" against those parameters:

```
// This is not legal C# 3.0
class Person(firstName, lastName, age : int)
{
    FirstName { get { firstName } set; }
    LastName { get { lastName } set; }
    Age { get { age } set; }
    FullName {
      get { return FirstName + " " + LastName; }
    }
}
```

Unfortunately, the syntax is getting tricky to parse, particularly if the constructor body is now implicitly "inside" the class. It's going to have difficulty knowing what denotes a member of the class and what denotes a local variable inside the constructor body. Even if the compiler could, the programmer may not, so an explicit declaration of what is a member and what isn't would be helpful:

```
// This is not legal C# 3.0
class Person(firstName, lastName, age : int)
{
    member FirstName { get { firstName } set; }
    member LastName { get { lastName } set; }
    member Age { get { age } set; }
    member FullName {
      get { return FirstName + " " + LastName; }
    }
}
```

If the compiler's going to infer properties, let it infer the default property implementation, a get/set pair against a field backdrop, and assign its first value:

```
// This is not legal C# 3.0
class Person(firstName, lastName, age : int)
{
    member FirstName = firstName;
    member LastName = lastName;
    member Age = age;
    member FullName = FirstName + " " + LastName;
}
```

Methods, of course, would have similar inferential treatment:

```
// This is not legal C# 3.0
class Person(firstName, lastName, age : int)
{
    member FirstName = firstName;
    member LastName = lastName;
    member Age = age;
    member FullName = FirstName + " " + LastName;
    override ToString() {
        return String.Format("{0} {1} {2}",
            FirstName, LastName, Age);
    };
}
```

Language-wide type inference is turning out to be quite the beneficial thing to have. Fortunately, the compiler is still fully aware of the types of each of these constructs, so static type safety remains viable. But if this compiler is going to continue to take burdens off the programmer's shoulders, then some kind of facility to address the burdens of concurrent-safe programming is necessary.

On top of all this, the language can start to make the explicit "generic" declarations of the earlier C# operations less necessary, because now the compiler will have the ability to infer the actual types of the parameters, and with it, the ability to infer them as generic type parameters:

```
var swap = delegate (l, r) {
    var temp = r; r = l; l = temp;
};
```

Here, the compiler can infer that `l` and `r` are of the same type, but that actual type is entirely irrelevant, because any type (class or struct, user-defined or BCL) can satisfy the inferred type parameters for `l` and `r`.

This would make much of the earlier code around currying so much, much easier to write and understand.

IMMUTABILITY

As an old joke goes, a man walks into a doctor's office and says, "Doctor, it hurts every time I do this" and jabs his thumb into his eye. The doctor, without missing a beat, says "Well, don't do that, and you'll be fine." Concurrency experts have long had a similar joke: If it hurts to write the locking code around every time the program changes state, then don't change state and you'll be fine. After all, if the variable never changes its state, then no update operation is possible and thus no locking code around those updates are necessary.

Although the proponents of fully-immutable variable state (also known as the "pure functional language") continue to wage loud and copious arguments against those who favor the merits of partially mutable variable state (the "impure language"), too many systems and libraries in the .NET ecosystem rely on the capability to change variable state in a running program to abandon the idea of mutable state entirely. That said, many objects in a .NET program remain immutable when initialized, and many other types could do so with little concern or change to their use:

```
Person talbott = new Person("Talbott", "Crowell", 29);
Person olderTalbott = new Person(talbott.FirstName,
    talbott.LastName, talbott.Age + 1);
```

Enforcing this, however, requires the developer writing the class type to ensure there's no way to modify the contents of those instances. This means that fields must be marked as read-only and properties with just a "get" handler.

If, however, the presumption is that most objects will remain unchanged when created, then rather than assuming the objects should be mutable by default, the language can make the opposite assumption and require the use of a keyword to indicate mutability.

Thanks to the power of inference, the programmer no longer has to stress over the low-level physical details of how the code projects itself onto the underlying CLR. Plus, perhaps surprisingly, none of the earlier syntax needs to change — the compiler simply chooses to generate the code differently, to be immutable by default instead of mutable.

Of course, now that the situation has reversed itself, if the programmer does want the ability to modify the internals of an object, the programmer must explicitly mark the parts of the class that need to be mutable:

```
// This is not legal C# 3.0
class Person(firstName, lastName, age : int)
{
    member FirstName = firstName;
    member LastName = lastName;
    mutable member Age = age;
    member FullName = FirstName + " " + LastName;
    override ToString() {
        return String.Format("{0} {1} {2}",
            FirstName, LastName, Age);
    };
}
```

This will not be enough to make the world safe for multiple threads of execution, but it will reduce the amount of thinking required to write thread-safe code.

EXPRESSIONS, NOT STATEMENTS

While the language syntax and semantics are up for discussion, an inconsistency within the traditional imperative language presents itself for possible correction: Certain constructs within the imperative language are expressions, yielding a result value, while other constructs are statements, yielding no value whatsoever. And in some languages, particularly those descending from the C++ wing of the language family tree, the ultimate inconsistency presents itself: Two different language constructs that do almost the same thing, except that one is a statement and the other an expression:

```
var anotherResult = false;
  if (x == 2)
      anotherResult = true;
  else
      anotherResult = false;
var yetAnotherResult = (x == 2) ? true : false;
```

The inconsistency is maddening. Particularly when it could be applied to a variety of other constructs — why doesn't switch/case have a similar kind of expression construct?

```
var thirdResult =
    switch (x)
    {
        case 0: "empty"; break;
        case 1: "one"; break;
        case 2: "two"; break;
        default: "many"; break;
    };
```

If every (or most every) language construct is an expression, it means the language takes a more input-yields-output style to it, which reinforces the general nature of testable programs, rather than just as a series of statements.

SUMMARY

This chapter began with a litany of the flaws of the object-oriented mindset and detailed what a new language might look like — one in which functions and methods were given first-class citizen status, the compiler could infer static type information from more of the language constructs, variable and field immutability serves as the default, and expressions formed the core of the language instead of imperative statements.

In short, we have just authored a language strikingly similar to the F# programming language. The remainder of this book details that language.